1. Neural Networks using Numpy

Assumptions:

1. We assumed that data passed into every function, x, was not modified after being returned from loadData(). Thus, we reshaped our data (training, validation or testing) in each function where necessary.

1.1. Helper Functions

1.1.1. $ReLU(): ReLU(x) = max(x, 0)$

```python
def relu(x):
    relu = np.maximum(x,0)
    return relu
```

Figure 1: Python Code for ReLU Function

1.1.2. $softmax(): \sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}, j = 1\ldots K, for\ K\ classes$

```python
def softmax(x):
  exp_z = np.exp(x)
  exp_sum = np.sum(exp_z, axis=1, keepdims=True)
  softmax = exp_z / exp_sum
  return softmax
```

Figure 2: Python Code for softmax Function

1.1.3. $computeLayet():$

```python
def computeLayer(X, W, b):
  compute_layer = np.matmul(X,W) + b
  return compute_layer
```

Figure 3: Python Code for computeLayer Function

1.1.4. $averageCE(): averageCE = -\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K} y_k^{\,n} log(p_k^{\,n})$

```python
def CE(target, prediction):
  N = len(target)
  average = -np.sum(target*np.log(prediction))/N
  return average
```

Figure 4: Python Code for CE Function

1.1.5. $gradCE()$

Prior to implementing the gradCE function in Python, we derived the gradient of cross entropy loss with respect to the input of softmax function.

$$L = -\sum_{k=1}^{K} y_k \log(p_k) = -y^T \log(p_k) = -y^T \log(\sigma)$$

$$p_k = softmax\ function\ output = \sigma(o)_j$$

$$\frac{\partial L}{\partial o} = \frac{\partial L}{\partial \sigma}\frac{\partial \sigma}{\partial o}$$
$$\frac{\partial L}{\partial \sigma} = -y^T \cdot \frac{1}{\sigma}$$
$$\frac{\partial \sigma}{\partial o} = \frac{\partial}{\partial o}\left(\frac{e^{\boldsymbol{o}}}{\sum_{k=1}^{K} e^{o_k}}\right) = \frac{e^{\boldsymbol{o}}\,(\sum_{k=1}^{K} e^{o_k}) - e^{\boldsymbol{o}}e^{\boldsymbol{o}}}{(\sum_{k=1}^{K} e^{o_k})^2} = \frac{e^{\boldsymbol{o}}\Big((\sum_{k=1}^{K} e^{o_k}) - e^{\boldsymbol{o}}\Big)}{(\sum_{k=1}^{K} e^{o_k})^2}$$
$$= \frac{e^{\boldsymbol{o}}}{(\sum_{k=1}^{K} e^{o_k})}\frac{\Big((\sum_{k=1}^{K} e^{o_k}) - e^{\boldsymbol{o}}\Big)}{(\sum_{k=1}^{K} e^{o_k})} = \frac{e^{\boldsymbol{o}}}{(\sum_{k=1}^{K} e^{o_k})}(1 - \frac{e^{\boldsymbol{o}}}{(\sum_{k=1}^{K} e^{o_k})})$$
$$= \sigma(1-\sigma)$$
$$\frac{\partial L}{\partial o} = \frac{\partial L}{\partial \sigma}\frac{\partial \sigma}{\partial o} = -y^T \cdot \frac{1}{\sigma} \cdot \sigma(1-\sigma) = \sigma - y$$

```python
def gradCE(target, o):
  gradient = softmax(o) - target
  return gradient
```

Figure 5: Python Code for gradCE Function

1.2. Backpropagation Derivation

The derivations below are implemented into Python code for backward propagation calculation in part 1.3.

1.2.1. $\frac{\partial L}{\partial W_o}$, *the gradient of the loss with respect to the outer layer weight*

$Shape: (K \times 10), with\ K\ units$

Prior to implementing the gradient function in Python, we derived the gradient of loss with respect to the outer layer weight on paper.

$$\frac{\partial L}{\partial W_o} = \frac{\partial L}{\partial p}\frac{\partial p}{\partial o}\frac{\partial o}{\partial W_o}$$
$$o = W_o h + b_o$$
$$\frac{\partial L}{\partial p} = -y^T \cdot \frac{1}{p}$$
$$\frac{\partial p}{\partial o} = \frac{\partial}{\partial o}\left(\frac{e^{\boldsymbol{o}}}{\sum_{k=1}^{K} e^{o_k}}\right) = \frac{e^{\boldsymbol{o}}\,(\sum_{k=1}^{K} e^{o_k}) - e^{\boldsymbol{o}}e^{\boldsymbol{o}}}{(\sum_{k=1}^{K} e^{o_k})^2} = \frac{e^{\boldsymbol{o}}\Big((\sum_{k=1}^{K} e^{o_k}) - e^{\boldsymbol{o}}\Big)}{(\sum_{k=1}^{K} e^{o_k})^2}$$
$$= \frac{e^{\boldsymbol{o}}}{(\sum_{k=1}^{K} e^{o_k})}\frac{\Big((\sum_{k=1}^{K} e^{o_k}) - e^{\boldsymbol{o}}\Big)}{(\sum_{k=1}^{K} e^{o_k})} = \frac{e^{\boldsymbol{o}}}{(\sum_{k=1}^{K} e^{o_k})}(1 - \frac{e^{\boldsymbol{o}}}{(\sum_{k=1}^{K} e^{o_k})})$$
$$= p(1-p)$$

$$\frac{\partial o}{\partial W_o} = h$$

$$\frac{\partial L}{\partial W_o} = \frac{\partial L}{\partial p}\frac{\partial p}{\partial o}\frac{\partial o}{\partial W_o} = h^T(p-y)$$

```
def dL_dWo(target, z, hidden_output):
  dL_do = gradCE(target, z) # N x 10
  do_dWo = hidden_output.transpose() # N x K
  dL_dWo = np.matmul(do_dWo,dL_do) # K x 10
  return dL_dWo
```

Figure 6: Python Code for Function that Calculates the Gradient of Loss With Respect to the Outer Layer Weight

1.2.2. $\frac{\partial L}{\partial b_o}$, *the gradient of the loss with respect to the outer layer bias*

$Shape: (1 \times 10)$

Prior to implementing the gradient function in Python, we derived the gradient of loss with respect to the outer layer bias on paper.

$$\frac{\partial L}{\partial b_o} = \frac{\partial L}{\partial p}\frac{\partial p}{\partial o}\frac{\partial o}{\partial b_o}$$

$$o = W_o h + b_o$$

$$\frac{\partial L}{\partial p} = -y^T \cdot \frac{1}{p}$$

$$\frac{\partial p}{\partial o} = \frac{\partial}{\partial o}\left(\frac{e^{\boldsymbol{o}}}{\sum_{k=1}^{K} e^{o_k}}\right) = \frac{e^{\boldsymbol{o}}\,(\sum_{k=1}^{K} e^{o_k}) - e^{\boldsymbol{o}}e^{\boldsymbol{o}}}{(\sum_{k=1}^{K} e^{o_k})^2} = \frac{e^{\boldsymbol{o}}\left((\sum_{k=1}^{K} e^{o_k}) - e^{\boldsymbol{o}}\right)}{(\sum_{k=1}^{K} e^{o_k})^2}$$

$$= \frac{e^{\boldsymbol{o}}}{(\sum_{k=1}^{K} e^{o_k})}\frac{\left((\sum_{k=1}^{K} e^{o_k}) - e^{\boldsymbol{o}}\right)}{(\sum_{k=1}^{K} e^{o_k})} = \frac{e^{\boldsymbol{o}}}{(\sum_{k=1}^{K} e^{o_k})}\left(1 - \frac{e^{\boldsymbol{o}}}{(\sum_{k=1}^{K} e^{o_k})}\right)$$

$$= p(1-p)$$

$$\frac{\partial o}{\partial b_o} = \mathbf{1}$$

$$\frac{\partial L}{\partial b_o} = \frac{\partial L}{\partial p}\frac{\partial p}{\partial o}\frac{\partial o}{\partial b_o} = \mathbf{1}^T(p-y)$$

```
def dL_dbo(target, z):
  dL_do = gradCE(target, z)
  do_dbo = np.ones((1,len(target)))
  dL_dbo = np.matmul(do_dbo, dL_do)
  return dL_dbo
```

Figure 7: Python Code for Function that Calculates the Gradient of Loss With Respect to the Outer Layer Bias

1.2.3. $\frac{\partial L}{\partial W_h}$, *the gradient of the loss with respect to the hidden layer weight*

$Shape: (F \times K), with\ F\ features,\ K\ units$

Prior to implementing the gradient function in Python, we derived the gradient of loss with respect to the hidden layer weight on paper.

$$\frac{\partial L}{\partial W_h} = \frac{\partial L}{\partial o}\frac{\partial o}{\partial h}\frac{\partial h}{\partial z}\frac{\partial z}{\partial W_h}$$

$$h = ReLU(W_h x + b_h), and\ let\ z = W_h x + b_h$$

$$\frac{\partial L}{\partial o} = p - y^T$$

$$\frac{\partial o}{\partial h} = W_o$$

$$\frac{\partial h}{\partial z} = \frac{\partial}{\partial z}\max(0, z) = \begin{cases} 0, & if\ z < 0 \\ 1, & if\ z > 0 \end{cases}$$

$$\frac{\partial z}{\partial W_h} = x$$

$$\frac{\partial L}{\partial W_h} = \frac{\partial L}{\partial p}\frac{\partial p}{\partial h}\frac{\partial h}{\partial z}\frac{\partial z}{\partial W_h} = x^T \otimes \frac{\partial h}{\partial z}(p - y^T)W_o^T$$

```
def dL_dWh(target, z, x, hidden_input, W_o):
  dL_do = gradCE(target, z)
  do_dh = W_o.transpose()
  dh_dz = relu_gradient(hidden_input)
  dz_dWh = x.transpose()
  dL_dh = np.matmul(dL_do,do_dh)
  dL_dz = dh_dz*dL_dh
  dL_dWh = np.matmul(dz_dWh,dL_dz)
  return dL_dWh
```

Figure 8: Python Code for Function that Calculates the Gradient of Loss With Respect to the Hidden Layer Weight

The above function makes use of additional helper function for getting derivatives of ReLU function.

```
def relu_gradient(x):
  return np.where(x < 0, 0, 1)
```

Figure 9: Python Code for Helper Function that Calculates the Gradient of ReLU Function

1.2.4. $\frac{\partial L}{\partial b_h}$, *the gradient of the loss with respect to the hidden layer bias*

$Shape: (1 \times K), with\ K\ units$

Prior to implementing the gradient function in Python, we derived the gradient of loss with respect to the hidden layer bias on paper.

$$\frac{\partial L}{\partial b_h} = \frac{\partial L}{\partial o}\frac{\partial o}{\partial h}\frac{\partial h}{\partial z}\frac{\partial z}{\partial b_h}$$

$$h = ReLU(W_h x + b_h), and\ let\ z = W_h x + b_h$$

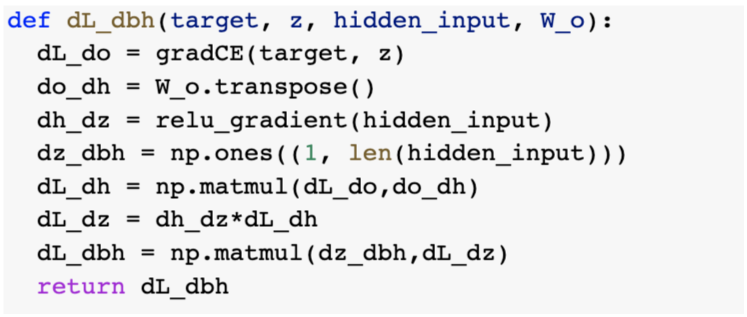
$$\frac{\partial L}{\partial o} = p - y^T$$

$$\frac{\partial o}{\partial h} = W_o$$

$$\frac{\partial h}{\partial z} = \frac{\partial}{\partial z}\max(0, z) = \begin{cases} 0, & if\ z < 0 \\ 1, & if\ z > 0 \end{cases}$$

$$\frac{\partial z}{\partial b_h} = \mathbf{1}$$

$$\frac{\partial L}{\partial W_h} = \frac{\partial L}{\partial p}\frac{\partial p}{\partial h}\frac{\partial h}{\partial z}\frac{\partial z}{\partial W_h} = \mathbf{1}^T \otimes \frac{\partial h}{\partial z}(p - y^T)W_o^T$$



```
def dL_dbh(target, z, hidden_input, W_o):
  dL_do = gradCE(target, z)
  do_dh = W_o.transpose()
  dh_dz = relu_gradient(hidden_input)
  dz_dbh = np.ones((1, len(hidden_input)))
  dL_dh = np.matmul(dL_do,do_dh)
  dL_dz = dh_dz*dL_dh
  dL_dbh = np.matmul(dz_dbh,dL_dz)
  return dL_dbh
```

Figure 10: Python Code for Function that Calculates the Gradient of Loss With Respect to the Hidden Layer Bias

1.3. Learning

The neural network in this part is constructed with a hidden unit size of 1000 that trains for 200 epochs. We assumed that train data are values directly returned by the loadData() and target data are one hot encoded. The weight matrices are initialized using Xaiver initialization scheme — hidden layer weight has zero-mean Gaussian distribution with variance of $\frac{2}{728+1000}$, output layer weight has zero-mean Gaussian distribution with variance of $\frac{2}{1000+10}$, and hidden layer bias and output layer bias are initialized to zero with respect to the shape of (1×1000) and (1×10)mentioned in the previous part. The learning rate of this neural network is $10^{-5}$and $\gamma$is 0.99.

Provided below is the implementation of backpropagation algorithm on neural network in Python based on the above derivations. The function below was modified to accommodate passing of training, validation and testing data for plotting graphs.

```python
def learning(train_data, train_target, val_data, val_target, test_data, test_target, epochs, hidden_units, gamma, alpha):

  # initialize data
  train_data = train_data.reshape([train_data.shape[0], train_data.shape[1]*train_data.shape[2]])
  val_data = val_data.reshape([val_data.shape[0], val_data.shape[1]*val_data.shape[2]])
  test_data = test_data.reshape([test_data.shape[0], test_data.shape[1]*test_data.shape[2]])

  # convert to onehot
  train_target, val_target, test_target = convertOneHot(train_target, val_target, test_target)

  # xavier initialization scheme
  variance_h = 2/(train_data.shape[0] + hidden_units)
  std_h = np.sqrt(variance_h)
  W_h = np.random.normal(0, std_h, (train_data.shape[1], hidden_units)) # 784 x 1000

  variance_o = 2/(hidden_units + 10)
  std_o = np.sqrt(variance_o)
  W_o = np.random.normal(0, std_o, (hidden_units, 10))

  b_h = np.zeros((1, hidden_units))
  b_o = np.zeros((1, 10))

  # initialize v to the size of W_h and W_o with value of 10^-5
  v_h = np.full((train_data.shape[1], hidden_units), 1e-5)
  v_o = np.full((hidden_units, 10), 1e-5)

  # plot initialization
  loss_train = []
  loss_valid = []
  loss_test = []

  accuracy_train = []
  accuracy_valid = []
  accuracy_test = []

  iterations = range(epochs)
```

```python
  for i in iterations:

    prediction, o, hidden_input, hidden_output = forward(train_data, W_h, b_h, W_o, b_o)
    cost_train = CE(train_target, prediction)
    loss_train.append(cost_train)

    acc_train = accuracy(train_target, prediction)
    accuracy_train.append(acc_train)

    prediction_valid, o_valid, hidden_input_valid, hidden_output_valid = forward(val_data, W_h, b_h, W_o, b_o)
    cost_valid = CE(val_target, prediction_valid)
    loss_valid.append(cost_valid)

    acc_valid = accuracy(val_target, prediction_valid)
    accuracy_valid.append(acc_valid)

    prediction_test, o_test, hidden_input_test, hidden_output_test = forward(test_data, W_h, b_h, W_o, b_o)
    cost_test = CE(test_target, prediction_test)
    loss_test.append(cost_test)

    acc_test = accuracy(test_target, prediction_test)
    accuracy_test.append(acc_test)

    # update W_h, W_o, b_h, b_o
    W_h, W_o, b_h, b_o = update(train_data, train_target, o, hidden_input, hidden_output, W_h, b_h, W_o, b_o, v_h, v_o, gamma, alpha)


  # plot loss
  plt.plot(iterations, loss_train)
  plt.plot(iterations, loss_valid)
  plt.plot(iterations, loss_test)
  plt.title("Neural Network Loss: hidden_units = 1000")
  plt.legend(['Training Loss','Validation Loss','Test Loss'])
  plt.xlabel("Epochs")
  plt.ylabel("Loss")
  plt.show()
```

```python
  # plot accuracy
  plt.plot(iterations, accuracy_train)
  plt.plot(iterations, accuracy_valid)
  plt.plot(iterations, accuracy_test)
  plt.title("Neural Network Accuracy: hidden_units = 1000")
  plt.legend(['Training Accuracy','Validation Accuracy','Test Accuracy'])
  plt.xlabel("Epochs")
  plt.ylabel("Accuracy")
  plt.show()
```

Figure 11: Python Code Neural Network Implementation

The above function uses additional helper functions to calculate forward propagation, backward propagation, and function that updates weights and bias.

```python
def forward(data, W_h, b_h, W_o, b_o):
  hidden_input = computeLayer(data, W_h, b_h)
  hidden_output = relu(hidden_input)
  o = computeLayer(hidden_output, W_o, b_o)
  prediction = softmax(o)
  return prediction, o, hidden_input, hidden_output
```

Figure 12: Python Code for Forward Propagation Calculation

```python
def update(data, target, prediction, hidden_input, hidden_output, W_h, b_h, W_o, b_o, v_h, v_o, gamma, alpha):

  v_w_h_new = gamma*v_h + alpha*dL_dWh(target, prediction, data, hidden_input, W_o)
  W_h_new = W_h - v_w_h_new

  v_w_o_new = gamma*v_o + alpha*dL_dWo(target, prediction, hidden_output)
  W_o_new = W_o - v_w_o_new

  v_b_h_new = gamma*b_h + alpha*dL_dbh(target, prediction, hidden_input, W_o)
  b_h_new = b_h - v_b_h_new

  v_b_o_new = gamma*b_o + alpha*dL_dbo(target, prediction)
  b_o_new = b_o - v_b_o_new

  return W_h_new, W_o_new, b_h_new, b_o_new
```

Figure 13: Python Code for Updating Weights and Bias

The above algorithm uses additional helper functions to plot accuracy. The helper function merely compares the difference between the prediction value and training target in order to plot accuracy as shown below.

```python
def accuracy(target, prediction):
  N = len(target)
  max_predict = np.argmax(prediction, axis=1)
  max_actual = np.argmax(target, axis=1)
  check = np.where(max_predict == max_actual, 1, 0)
  accuracy = np.sum(check)/N

  return accuracy
```

Figure 14: Python Code for Helper Function that Plots Accuracy

The final accuracy for this algorithm is:

Training Data Accuracy	Validation Data Accuracy	Test Data Accuracy
90.4%	89.35%	89.50%

Table 1: Summary of Training, Validation, and Test Accuracy with Hidden Unit Size of 1000

Given below are plots of loss and accuracy for training, validation, and test data set.

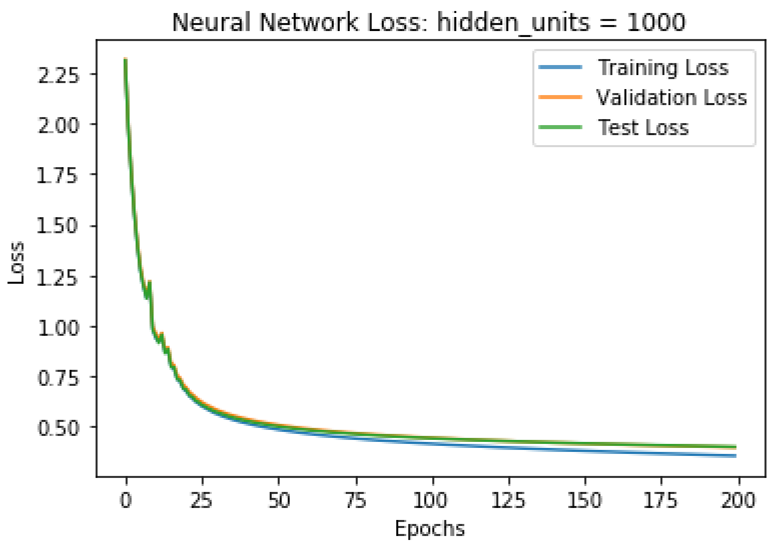


Figure 15: Training, Validation, and Test Accuracy Plot with Hidden Unit Size of 1000

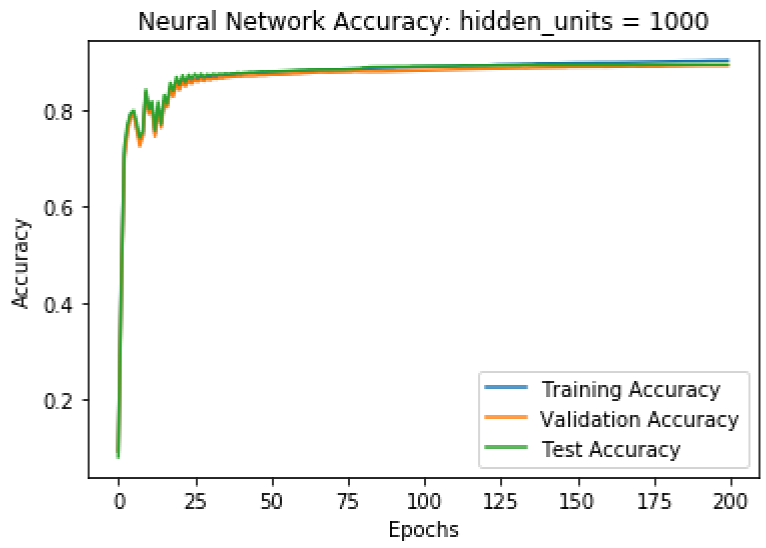


Figure 16: Training, Validation, and Test Accuracy Plot with Hidden Unit Size of 1000

1.4. Hyperparameter Investigation

1.4.1. Number of hidden units

Given below are the results for investigating the impact of modifying the hidden unit size. In all the instances, we have maintained learning rate = 0.99 and 200 epochs.

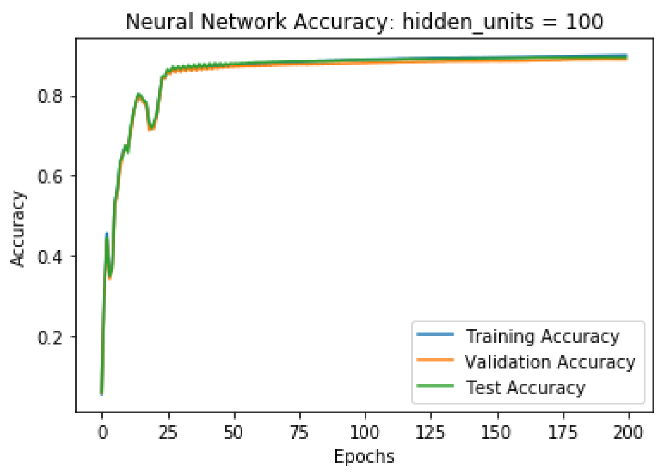


Figure 17: Training, Validation, and Test Accuracy Plot with Hidden Unit Size of 100

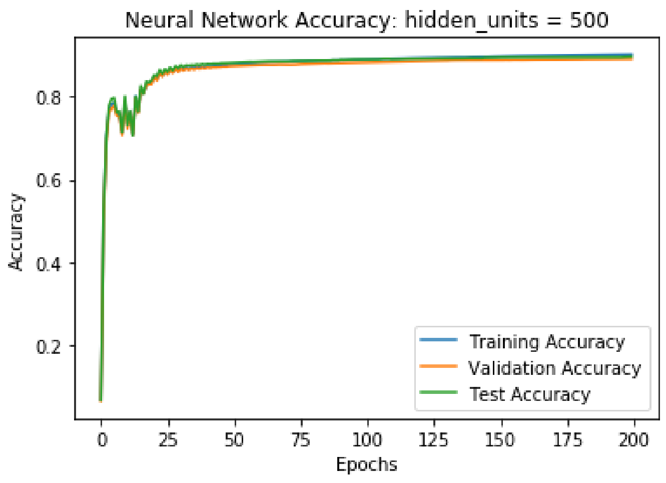


Figure 18: Training, Validation, and Test Accuracy Plot with Hidden Unit Size of 500

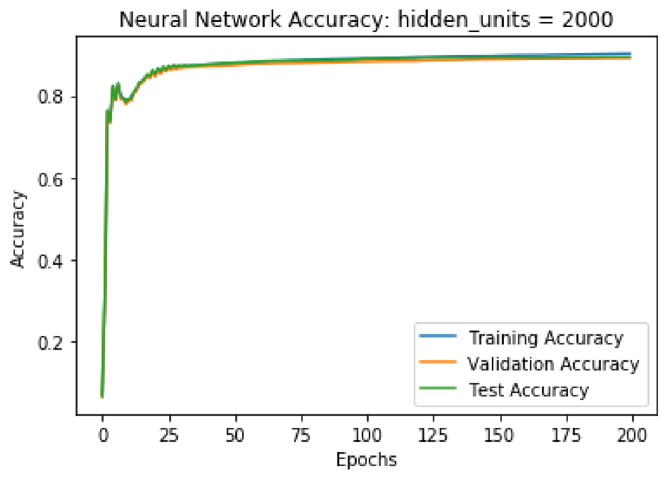


Figure 19: Training, Validation, and Test Accuracy Plot with Hidden Unit Size of 2000

Below is a table of comparison of training, validation, and test data accuracy over hidden unit size of 100, 500, and 2000.

Hidden Unit Size	Training Data Accuracy	Validation Data Accuracy	Test Data Accuracy
100	89.97%	89.00%	89.46%
500	90.14%	89.00%	89.61%
2000	90.35%	89.23%	89.60%

Table 2: Summary of Training, Validation, and Test Accuracy with Hidden Unit Size of 100, 500, and 2000

From the table above, we observe that the larger hidden unit size leads to higher accuracy. Furthermore, larger hidden unit size also leads to longer training time.

From the data above it is noticeable that the training, validation, and test accuracy slightly increase as the number of hidden units in the model increase from 100 to 2000. Further, it was observable that as the number of hidden units increased, the training time was longer. Going from 500 to 2000, there was a very high increase in training time, even though the change in test data accuracy was very low. It can be said that at around 500 hidden units, the model offers both high accuracy and performance.

1.4.2. Early Stopping

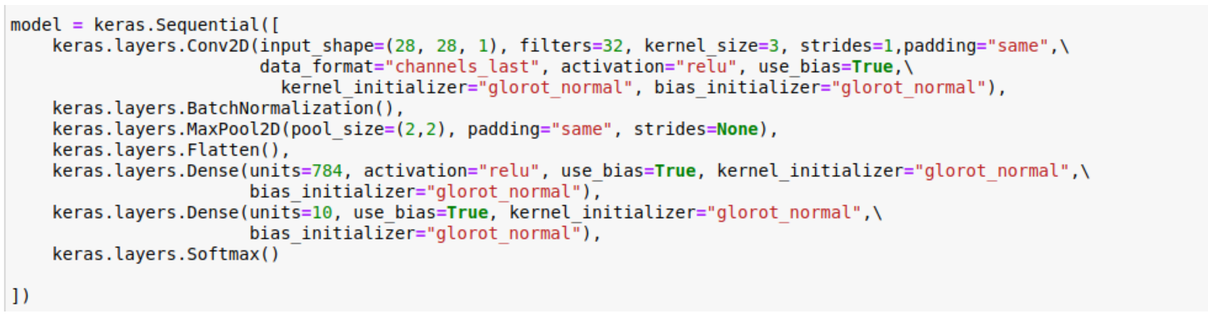
Early stopping is a technique used to avoid overfitting of the model to the training data. Generally, early stopping point can be found when the accuracy of validation data reaches its highest value and begins to decrease or plateau thereafter. Based on our graphs for 1000 hidden units and 200 epochs, overfitting did not seem to occur, as there is no point (epoch) where the validation loss starts to increase, despite decreasing training loss, and validation accuracy continues to grow even after 200 epochs. So, based on the plots in Section 1.3, the early stopping point would be 200 epochs (maximum epochs). The training accuracy is 90.4%, validation accuracy is 89.35%, and the test accuracy is 89.50%, which are same values as stated in Table 1.

2. Neural Networks in Tensorflow

Note: The following convolutional neural network implementation is based on tensorflow 2.x.

2.1. Model Implementation

Please note that the following implementation of convolutional neural network makes use of the "Keras" Tensorflow library. This model achieves the same specifications as outlined in the assignment handout.

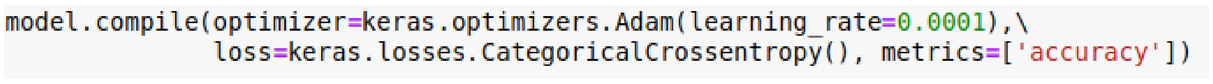


```
model = keras.Sequential([
    keras.layers.Conv2D(input_shape=(28, 28, 1), filters=32, kernel_size=3, strides=1,padding="same",\
                        data_format="channels_last", activation="relu", use_bias=True,\
                          kernel_initializer="glorot_normal", bias_initializer="glorot_normal"),
    keras.layers.BatchNormalization(),
    keras.layers.MaxPool2D(pool_size=(2,2), padding="same", strides=None),
    keras.layers.Flatten(),
    keras.layers.Dense(units=784, activation="relu", use_bias=True, kernel_initializer="glorot_normal",\
                       bias_initializer="glorot_normal"),
    keras.layers.Dense(units=10, use_bias=True, kernel_initializer="glorot_normal",\
                       bias_initializer="glorot_normal"),
    keras.layers.Softmax()

])
```

Figure 20. Convolutional network coded using Keras Tensorflow

Cross Entropy error is implemented while compiling the model using the following code,



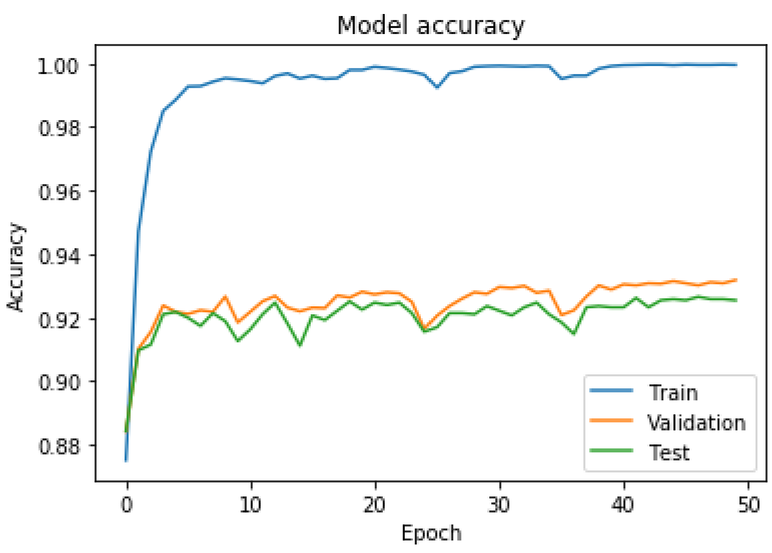
```
model.compile(optimizer=keras.optimizers.Adam(learning_rate=0.0001),\
              loss=keras.losses.CategoricalCrossentropy(), metrics=['accuracy'])
```

Figure 21. Adam optimizer parameters and Cross entropy loss function used for model

Our CNN solves a multi-class classification problem, thus, we make use of "CategoricalCrossEntropy" loss function.

2.2. Model Training

Following are the plots obtained from training the above model for 50 epochs, Adam optimizer with learning rate = 0.0001, training data shuffling on, and batch size of 32.



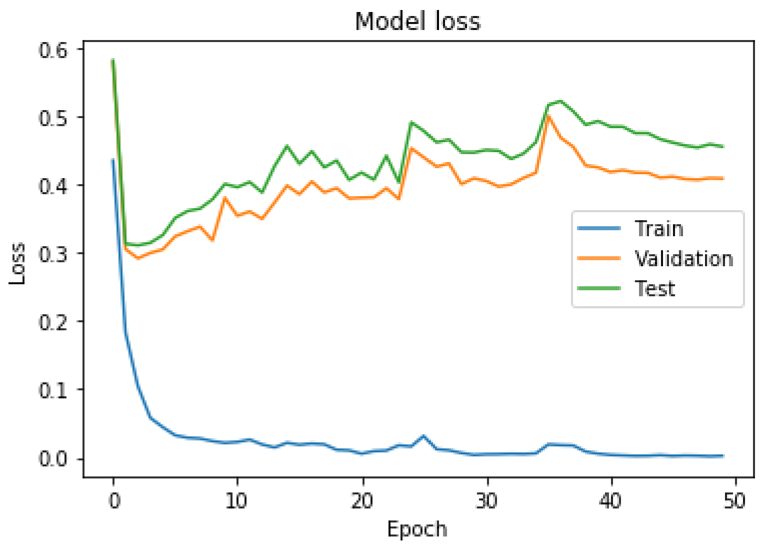


Figure 22a. Training, Validation and Test accuracies after training with with stated specifications
Figure 22b. Training, Validation and Test losses after training with stated specifications

2.3. Hyperparameter Investigation

1. L2 Regularization

The following results are obtained by adding regularization on the fully-connected layers of the CNN. All other parameters for training the model are unchanged from part 2.2 above.

Weight Decay Coefficients	Final training accuracy	Final validation accuracy	Final test accuracy
0.01	0.9851999878883362	0.9303333163261414	0.9284141063690186
0.1	0.930899977684021	0.9193333387374878	0.9240087866783142
0.5	0.883899986743927	0.8896666765213013	0.8924375772476196

Table 3. Summary of accuracies after adding L2 regularization to CNN

Weight Decay Coefficients	Final training loss	Final validation loss	Final test loss
0.01	0.18600476067066193	0.358395014444987	0.3737385141254347
0.1	0.4915312662124634	0.5267671669324239	0.5189385828117618
0.5	0.9747252080917358	0.9804146107037862	0.9738260946960001

Table 4. Summary of losses after adding L2 regularization to CNN

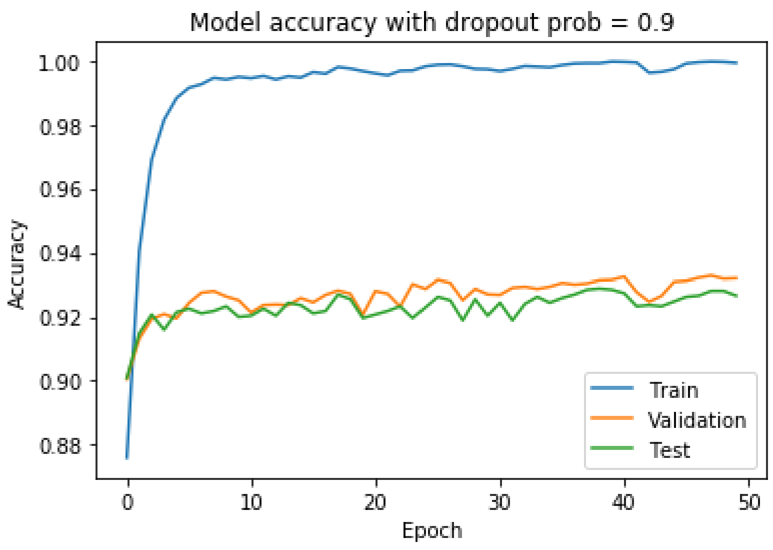
It can be observed that:

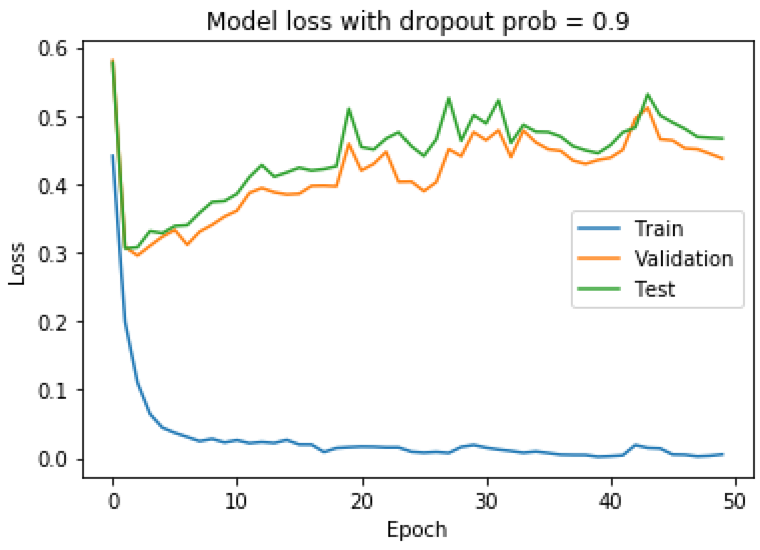
1. With increasing weight decay coefficients, final validation and test accuracies decrease.
2. With increasing weight decay coefficients, final validation and test losses increase.

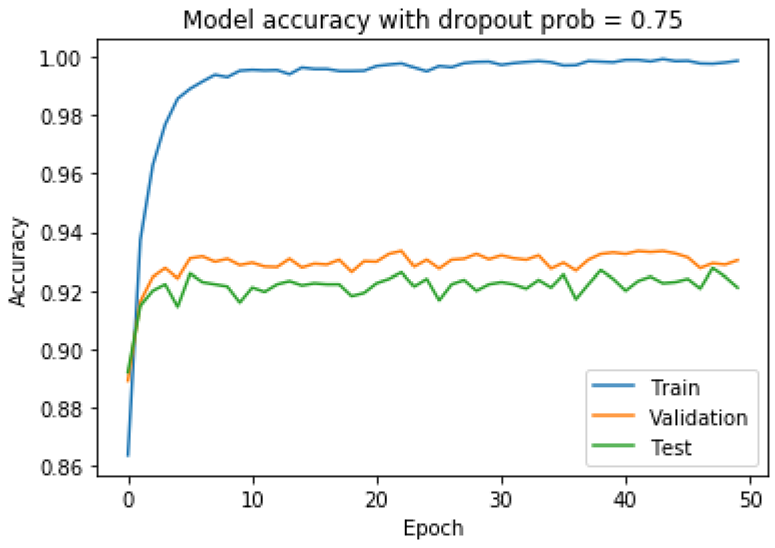
2. Dropout

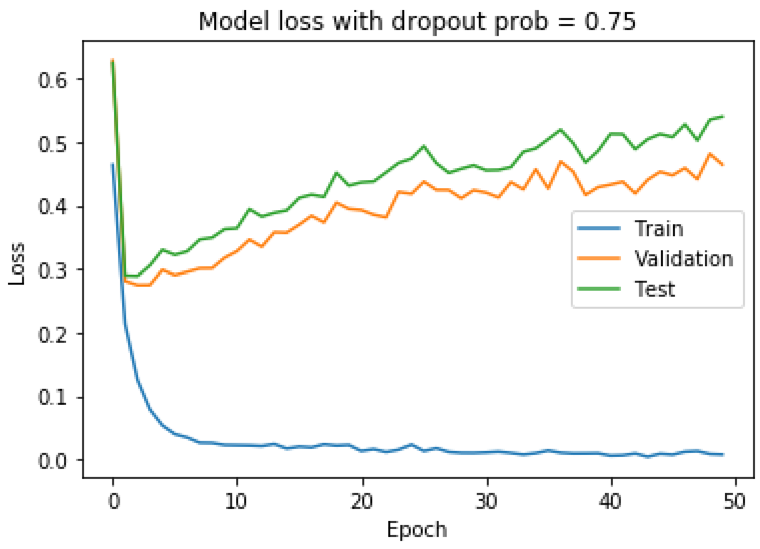
For the following plots, dropout layer is added after first “Dense” layer from figure 20. There is no regularization and all parameters are unchanged from section 2.2 above.

Titles in the plots below indicate dropout probabilities used.

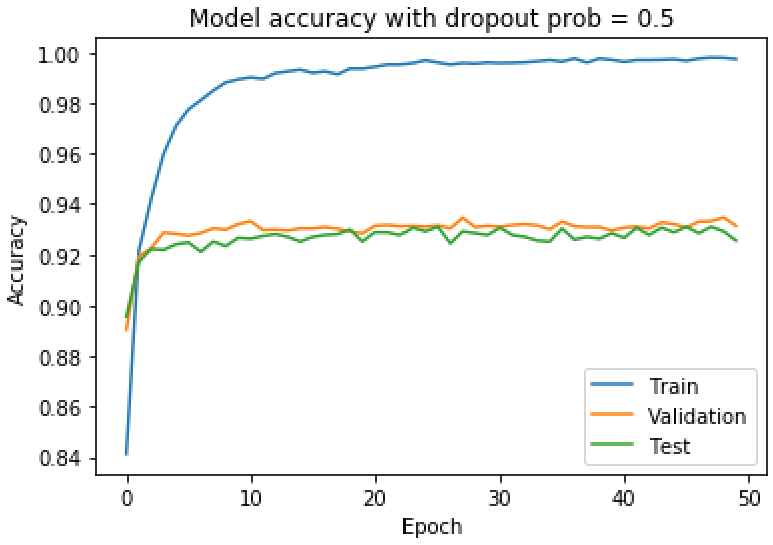

Model accuracy with dropout prob = 0.9
Accuracy
Epoch
Train
Validation
Test
1.00
0.98
0.96
0.94
0.92
0.90
0.88
0
10
20
30
40
50


Model loss with dropout prob = 0.9
Loss
Epoch
Train
Validation
Test
0.6
0.5
0.4
0.3
0.2
0.1
0.0
0
10
20
30
40
50

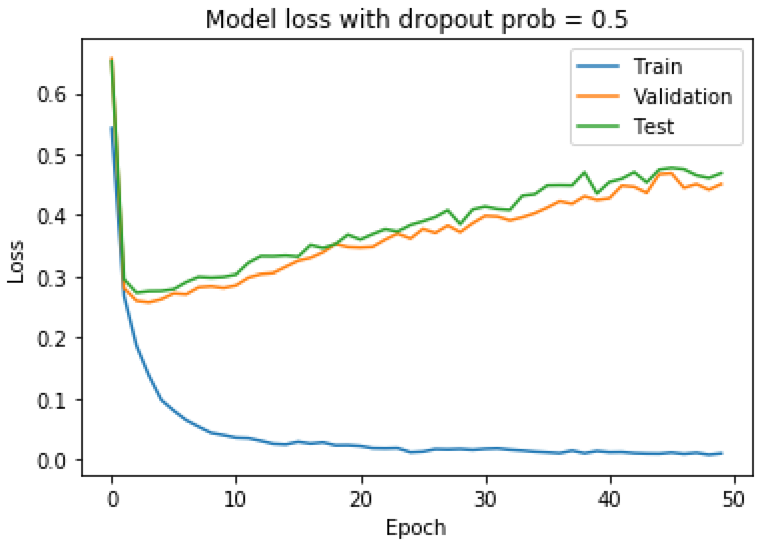

Model accuracy with dropout prob = 0.75
Accuracy
Epoch
Train
Validation
Test
1.00
0.98
0.96
0.94
0.92
0.90
0.88
0.86
0
10
20
30
40
50



Model loss with dropout prob = 0.75
Loss
Epoch
Train
Validation
Test
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0
10
20
30
40
50



Model accuracy with dropout prob = 0.5
Accuracy
Epoch
Train
Validation
Test
0.84
0.86
0.88
0.90
0.92
0.94
0.96
0.98
1.00
0
10
20
30
40
50



Model loss with dropout prob = 0.5
Loss
Epoch
Train
Validation
Test
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0
10
20
30
40
50